

Chapter 3: Processes

To introduce the notion of a process

To describe the various features of processes

To explore interprocess communication

To describe communication in client-server systems



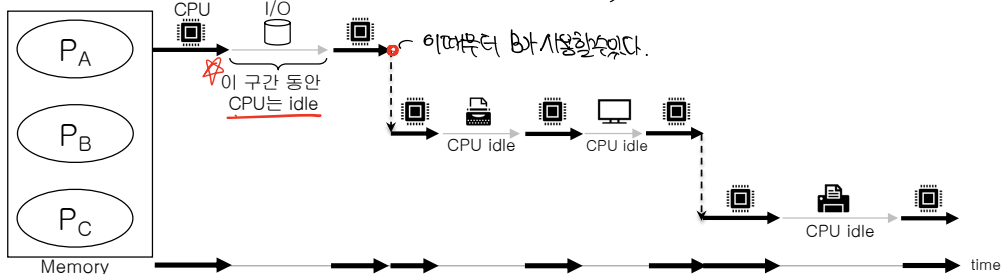
Contents

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

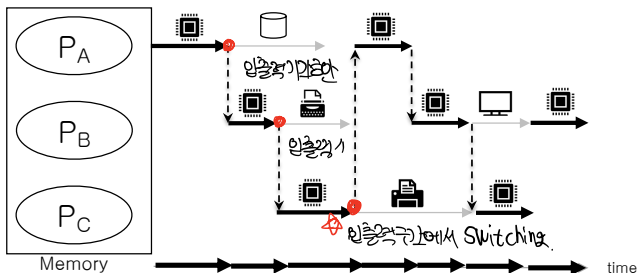


Multiprogramming

Uni-programming (한번에 하나의 프로그램만 실행)



Multi-programming (다수의 프로그램이 동시에 실행)



Throughput ↑

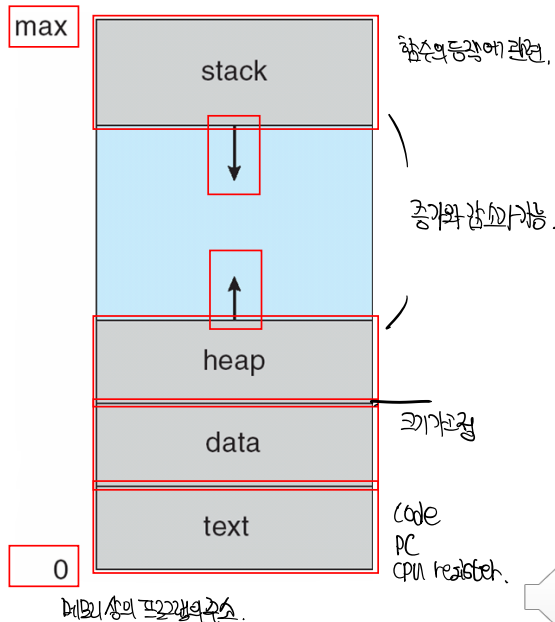
Utilization ↑

Degree of multiprogramming



3.1 Process Concept

- An operating system executes a variety of programs:
 - Batch system – **jobs** 실행해야 하는 프로그램
 - Time-shared systems – **user programs** or **tasks**
 - Textbook uses the terms **job** and **process** almost interchangeably
- **Process** – a program in execution
- Multiple parts
 - The program code, also called **text section**
 - Current activity, including **program counter**, processor registers
 - **Stack** containing temporary data
 - Function parameters, return addresses, local variables
 - **Data section** containing global variables
 - **Heap** containing memory dynamically allocated during run time



Process Concept (Cont.)

- Program is **passive** entity stored on disk (executable file), process is **active**
 - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
 - Consider multiple users executing the same program



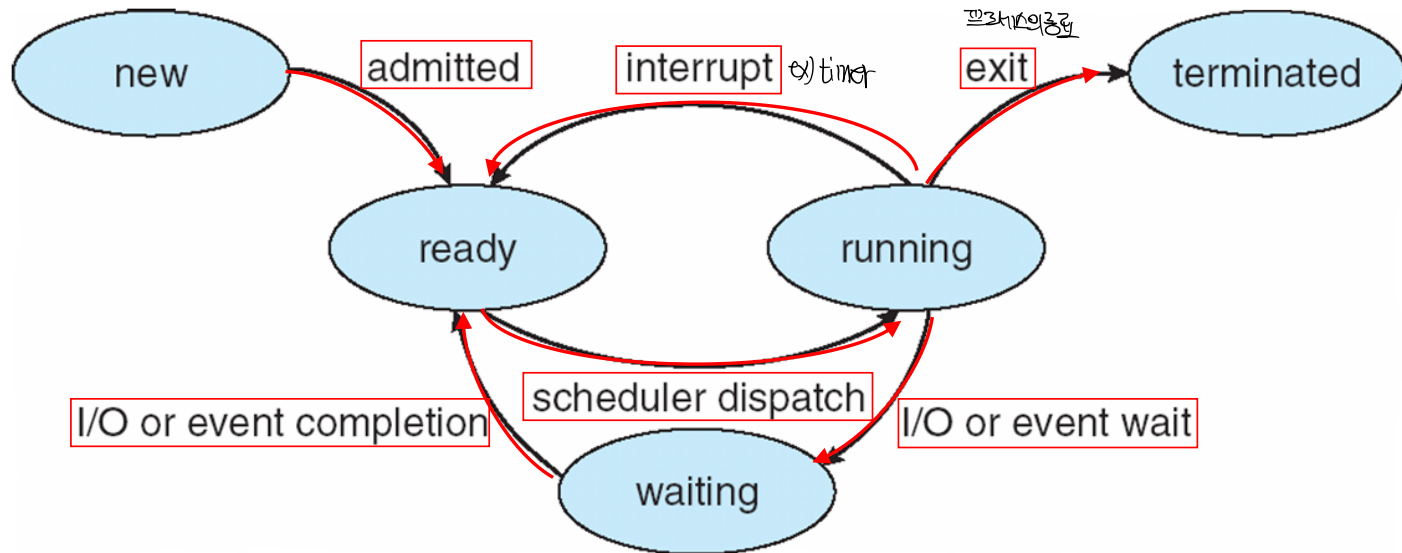
Process State

- As a process executes, it changes **state**
 - **new**: The process is being created
 - **ready**: The process is waiting to be assigned to a processor 실행될수있는 상태
 - **running**: Instructions are being executed
 - **waiting**: The process is waiting for some event to occur ex) I/O의완전
 - **terminated**: The process has finished execution



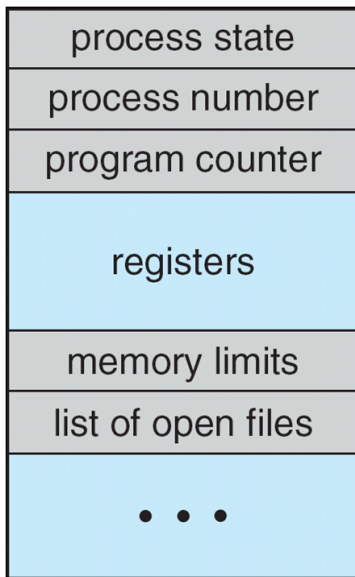
Diagram of Process State

⇒ 프로그램 상태 변화 다이어그램



✧ Process Control Block (PCB)

= 데이터이 개별프로세스를 위한 자료구조.



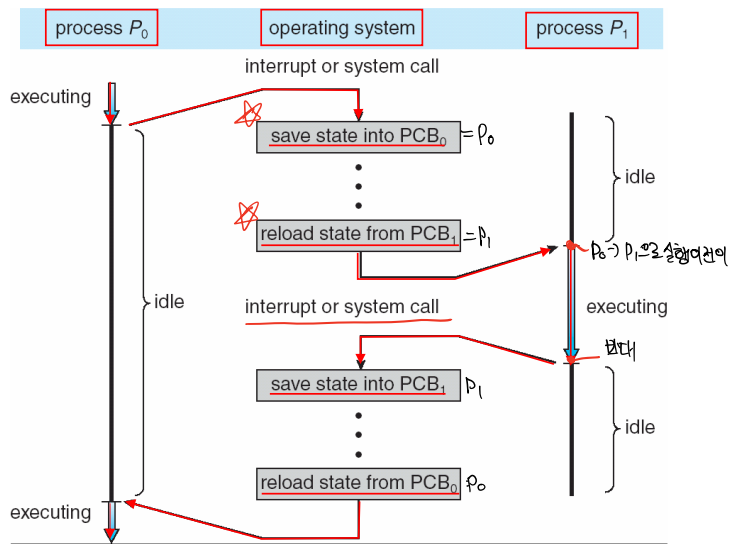
Information associated with each process(also called **task control block**)

- Process state – running, waiting, etc
- Process ID
- Program counter – location of instruction to next execute 복사본이다.
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits 예) 파크에서 사용.
- I/O status information – I/O devices allocated to process, list of open files



CPU Switch From Process to Process

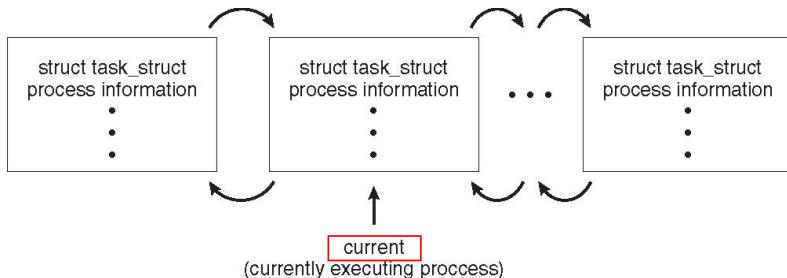
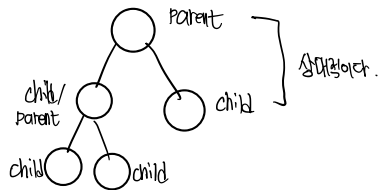
=> CPU 사용의 전환



Process Representation in Linux

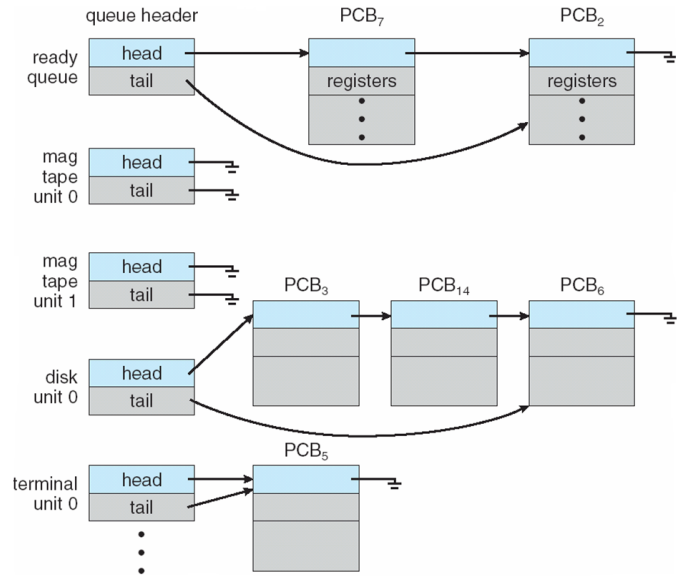
Represented by the C structure `task_struct`

```
pid_t pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



3.2 Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
⇒ 가용한 프로세스를 선택한다.
- Maintains **scheduling queues** of processes
 - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
 - **Device queues** – set of processes waiting for an I/O device
 - **Job queue** – set of all processes in the system
- Processes migrate among the various queues



Schedulers

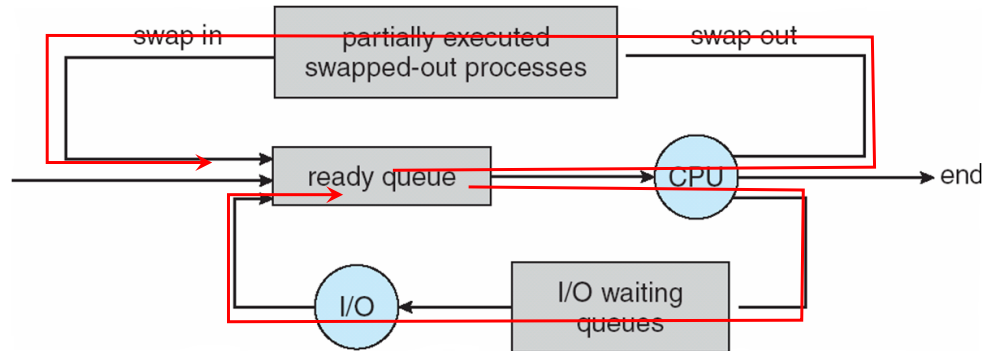
job을 선택하는 역할.

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU ⇒ ready queue에 대해 동작.
 - Sometimes the only scheduler in a system
 - Short-term scheduler is **invoked frequently** (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue ⇒ 작업을 ready queue에 올리는 역할.
 - Long-term scheduler is **invoked infrequently** (seconds, minutes) ⇒ (may be slow)
 - The long-term scheduler controls the **degree of multiprogramming** ⇒ 조정하는 역할.
- Processes can be described as either: ⇒ 그와 CPU bound의 속을 조절한다.
 - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
 - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good **process mix**

프로세스의 성향

Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
 - Remove process from memory, store on disk, bring back in from disk to continue execution: swapping 디스크의 메모리 공간으로 (바이트단) 이동가능해서 다시 올.



Context Switch

⇒ 상태 전환 기의 동작을 의미 ($P_0 \rightarrow P_1 / P_1 \rightarrow P_0$)

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch** 외부에서 관리하는 것 같다.
 1. register
 2. memory
 3. system.
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
 - The more complex the OS and the PCB → the longer the context switch ⇒ context switch 자체가 overhead 이다.
- Time dependent on hardware support
 - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once



3.3 Operations on Processes

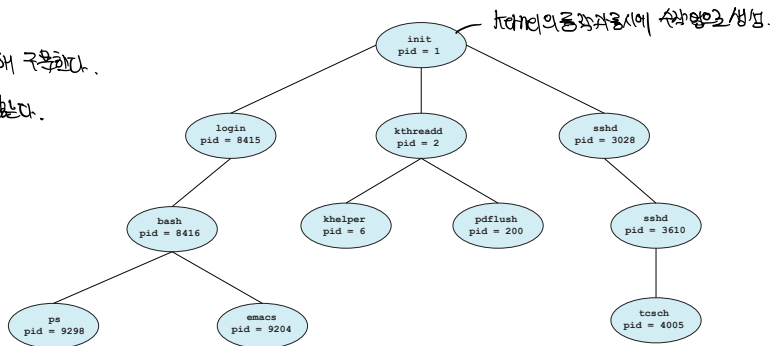
- System must provide mechanisms for: 대표공인연사
System 예들 통해 소개가능.
 - process creation
 - process termination
 - and so on as detailed next



Process Creation

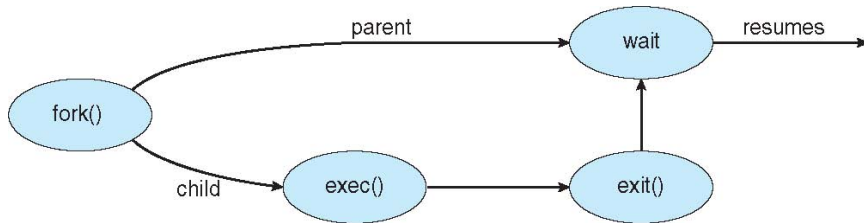
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier (pid)** 이름에 구별된다.
- ✗ **Resource sharing options** 1. 공유한다. 2. 공유하지 않는다.
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- **Execution options**
 - Parent and children execute concurrently
 - Parent waits until children terminate

linux의 process tree.



Process Creation (Cont.)

- Address space
 - Child duplicate of parent
 - Child has a program loaded into it
- UNIX examples
 - **fork()** system call creates new process ★ 중복과 동일한 프로세스의 이미지 생성.
 - **exec()** system call used after a **fork()** to replace the process' memory space with a new program 이미지를 교체해 자신의 프로그램으로 생성.



C Program Forking Separate Process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
```

```
int main()
{
    pid_t pid;
```

```
    /* fork a child process */
    pid = fork();
```

```
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
```

```
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
```

```
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }
```

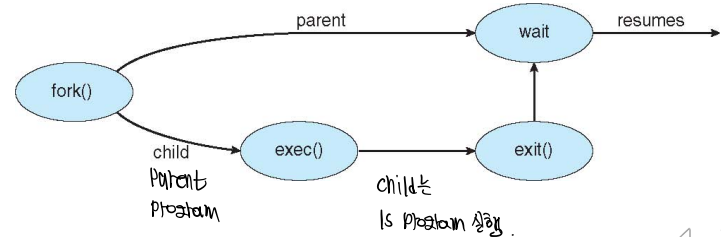
```
    return 0;
}
```

(Unix의 fork)

switch 나 if으로 구분하지 않음.

부모는 자식의 PID

자식은 0 PID를 리턴



Creating a Separate Process via Windows API

= Win32 API

```
#include <stdio.h>
#include <windows.h>
```

```
int main(VOID)
```

```
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
```

```
    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));
```

```
    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
        "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
        NULL, /* don't inherit process handle */
        NULL, /* don't inherit thread handle */
        FALSE, /* disable handle inheritance */
        0, /* no creation flags */
        NULL, /* use parent's environment block */
        NULL, /* use parent's existing directory */
        &si,
        &pi))
```

```
{
    fprintf(stderr, "Create Process Failed");
    return -1;
}
/* parent will wait for the child to complete */
WaitForSingleObject(pi.hProcess, INFINITE);
printf("Child Complete");

/* close handles */
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
}
```



Process Termination



- Process executes last statement and then asks the operating system to delete it using the **exit()** system call.
 - Returns status data from child to parent (via **wait()**)
 - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **abort()** system call. Some reasons for doing so:
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required
 - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination

- Some operating systems do not allow child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
 - **cascading termination.** All children, grandchildren, etc. are terminated.
 - The termination is initiated by the operating system. 자신의 하위 프로세스 모두 종료.
- The parent process may wait for termination of a child process by using the **wait()** system call. The call returns status information and the pid of the terminated process
`pid = wait(&status);` unix의 wait 함수 / exit(n)을 통한 종료 상태를 리턴.
- If no parent waiting (did not invoke **wait()**) process is a **zombie**
- If parent terminated without invoking **wait**, process is an **orphan**

3.4 Interprocess Communication

⇒ IPC

- Processes within a system may be **independent** or **cooperating**
- Reasons for cooperating processes:
 - Information sharing
 - Computation speedup
 - Modularity
 - Convenience
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
 -  **Shared memory**
 -  **Message passing**

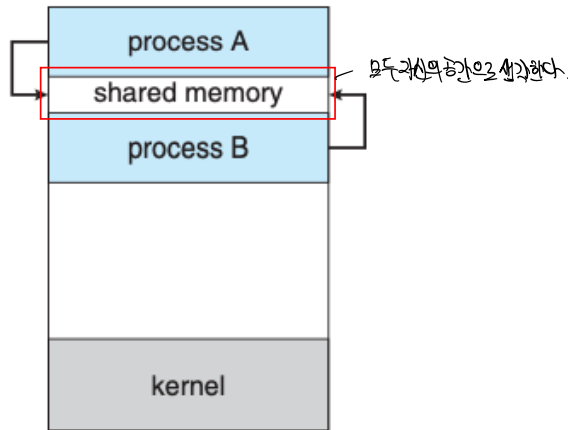
독립적인 동작
⇒ 영향 X

협력하는 동작
⇒ 영향 받는다.

Interprocess Communication – Shared Memory

일국 메모리 공간을 공유.

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory. *Synchronization is discussed in great details in Chapter 5. OS에게 제공하라.



Producer-Consumer Problem

= synchronize 의 문제의 예시

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
 - **unbounded-buffer** places no practical limit on the size of the buffer
 - **bounded-buffer** assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10 버퍼 크기
```

```
typedef struct {
```

```
    . . .
```

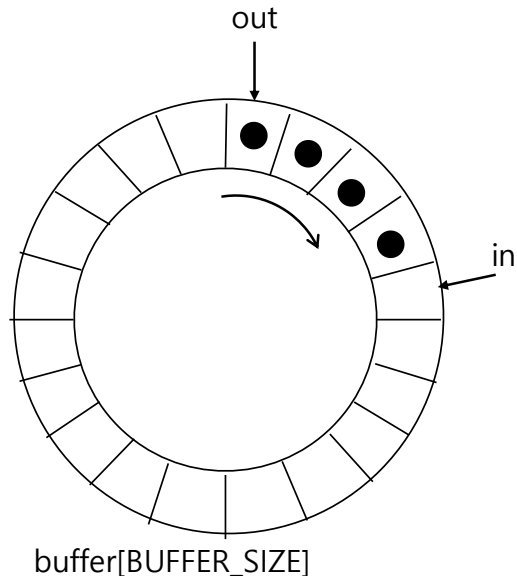
```
} item;
```

```
item buffer[BUFFER_SIZE];
```

```
int in = 0;
```

```
int out = 0;
```

Buffer의 위치를 가리키는 pointer 역할.



Bounded-Buffer – Producer and Consumer

Producer

item next_produced;

```
while (true) {
```

```
    /* produce an item */
```

```
    *while (((in+1)%BUFFER_SIZE)==out)
```

```
        ; /* do nothing */
```

```
    buffer[in] = next_produced;
```

```
    in = (in + 1) % BUFFER_SIZE;
```

```
}
```

Using Buffer

Consumer

item next_consumed;

```
while (true) {
```

```
    *while (in == out)
```

```
        ; /* do nothing */
```

```
    next_consumed = buffer[out]; /* do nothing */
```

```
    out = (out + 1) % BUFFER_SIZE;
```

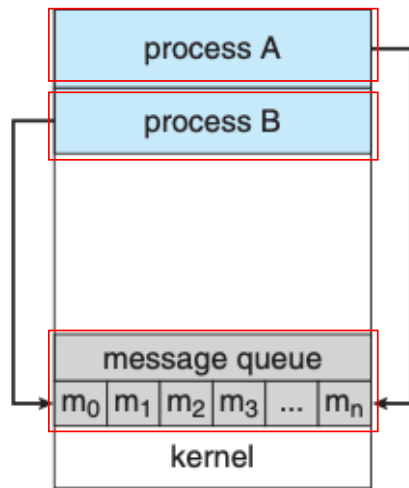
```
    /* consume the item */
```

```
}
```



Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
 - **send**(*message*)
 - **receive**(*message*)
- The *message* size is either fixed or variable



Direct Communication

- Processes must name each other explicitly:
 - **send** (P *message*) – send a message to process P
 - **receive**(Q *message*) – receive a message from process Q
- Properties of communication link
 - Links are established automatically
 - A link is associated with exactly one pair of communicating processes. Between each pair there exists exactly one link
 - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
 - Each mailbox has a unique id
 - Processes can communicate only if they share a mailbox
- Properties of communication link
 - Link established only if processes share a common mailbox
 - A link may be associated with many processes
 - Each pair of processes may share several communication links
 - Link may be unidirectional or bi-directional

Indirect Communication

- Operations
 - create a new mailbox (port)
 - send and receive messages through mailbox
 - destroy a mailbox
- Primitives are defined as:
 - send**(A , *message*) – send a message to mailbox A
 - receive**(A , *message*) – receive a message from mailbox A
- Mailbox sharing
 - P_1 , P_2 and P_3 share mailbox A
 - P_1 sends; P_2 and P_3 receive
 - Who gets the message?
 - Allow a link to be associated with at most two processes
 - Allow only one process at a time to execute a receive operation
 - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
 - **Blocking send** -- the sender is blocked until the message is received
 - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
 - **Non-blocking send** -- the sender sends the message and continue
 - **Non-blocking receive** -- the receiver receives:
 - A valid message, or
 - Null message
- Different combinations possible
 - If both send and receive are blocking, we have a rendezvous



Synchronization (Cont.)

- Producer-consumer becomes trivial

```
message next_produced;
while (true) {
    /* produce an item in next produced */
    send(next_produced);
}

message next_consumed;
while (true) {
    receive(next_consumed);

    /* consume the item in next consumed */
}
```



Buffering

- Queue of messages attached to the link.
- implemented in one of three ways
 1. Zero capacity – no messages are queued on a link.
Sender must wait for receiver (rendezvous)
 2. Bounded capacity – finite length of n messages
Sender must wait if link full
 3. Unbounded capacity – infinite length
Sender never waits

